

Atty. Docket No. MS303259.2/MSFTP447USA


CONTRACTS AND FUTURES IN AN
ASYNCHRONOUS PROGRAMMING LANGUAGE

by

James R. Larus, Sriram K. Rajamani, and Jakob Rehof

MAIL CERTIFICATION

I hereby certify that the attached patent application (along with any other paper referred to as being attached or enclosed) is being deposited with the United States Postal Service on this date September 10, 2003, in an envelope as "Express Mail Post Office to Addressee" Mailing Label Number EV330021166US addressed to the Mail Stop Patent Application, Commissioner for Patents, P.O. Box 1450, Alexandria, Virginia 22313-1450.


Eric D. Jorgenson

Title: CONTRACTS AND FUTURES IN AN ASYNCHRONOUS PROGRAMMING
LANGUAGE

5

CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims priority from pending U.S. Provisional Patent Application Ser. No. 60/450,982 entitled "CONTRACTS AND FUTURES IN AN ASYNCHRONOUS PROGRAMMING LANGUAGE" filed February 28, 2003.

10

TECHNICAL FIELD

This invention is related to an alternative concurrent programming model in computers, and more specifically, a programming language for facilitating such asynchronous concurrent operations.

15

BACKGROUND OF THE INVENTION

Programming languages, for the most part, impose a synchronous view on an increasingly asynchronous world. In these languages, operations are totally ordered, so that, given a program's input data, the entire sequence of its computation is determined. However, for better or worse, computation is becoming increasingly asynchronous. At the lowest level, architectural innovation is increasingly focused on improving performance through increased concurrency, with approaches such as simultaneous multithreading or single-chip multiprocessors. Equally significantly, ubiquitous networks make collaboration among programs and users possible, but parallelism and communication introduce asynchrony.

25

Programming languages, despite the importance of asynchrony in reactive, event-driven, concurrent, or parallel applications, typically support asynchrony only through library mechanisms, such as threads, events, and synchronization, and provide no linguistic or error-checking features. Lack of support is a serious problem, as asynchrony makes the difficult task of programming even harder, since partially ordered behavior is harder to both understand and analyze and because nondeterminism obscures the cause of errors.

30

Thus what is needed is an improved asynchronous programming language, system or methodology that mitigates some of the aforementioned shortcomings of conventional languages.

5

SUMMARY OF THE INVENTION

The following presents a simplified summary of the invention in order to provide a basic understanding of some aspects of the invention. This summary is not an extensive overview of the invention. It is not intended to identify key/critical elements of the invention or to delineate the scope of the invention. Its sole purpose is to present some concepts of the invention in a simplified form as a prelude to the more detailed description that is presented later.

The present invention discloses, in one aspect thereof, a novel programming language extension for expressing asynchronous computations, which combines the well-known ideas of futures and joins. A programmer is now permitted to write interface contracts that prescribe the ordering and overlap of a collection of asynchronous functions. The language supports asynchronous programming by way of additional programming constructs and a behavioral type system that statically detects common errors. The language is an extension of a C# language that supports an alternative concurrent programming model based on futures, asynchronous function calls, and joins on futures, which are building blocks for asynchronous systems.

One aspect of the present invention provides that, on any given client and service interface, a novel technique for automatically extracting behavioral models of both the client interface and the service interface. The disclosed programming language is designed for modular checking where contracts are specified on interface boundaries, interface clients and providers (or services) are checked independently, and each method is checked separately. A client model and a service model are automatically extracted from the respective interfaces to arrive at the minimum number of relevant descriptions, which are the possible behavioral aspects that govern communicative interaction with respect to each other. This is performed on an individual function level, in contrast to conventional systems where this may be performed with much difficulty on the program as a whole. The client and server models are then passed to a model checker to

systematically explore all of the possible interactions between the client and server code, including delays, reordering, etc., and to resolve ways in which to react to such possible interactions. Thus behavioral contracts can be stated on the asynchronous interfaces that prescribe the ordering and overlap of a collection of asynchronous functions.

5 A conformance checker uses the contracts to check that clients of the interface are invoked in the appropriate order and handle both normal and exceptional results, and the any interface client obeys the interface contract. The checker also checks that the interface's provider (or service) properly implements the contract. The checking algorithm combines region based type systems with model checking techniques to handle
10 pointer aliasing in a sound way.

One aspect of the present invention provides a novel technique for automatically extracting behavioral models of both the client and the service that are then fed to a model checker. The model checker relies on a new application of region types, where regions are interpreted as equivalence classes of modeled objects. The extracted models
15 are sound in the presence of aliasing, because the model checker interprets every object reference by nondeterministically choosing an object from its region.

To the accomplishment of the foregoing and related ends, certain illustrative aspects of the invention are described herein in connection with the following description and the annexed drawings. These aspects are indicative, however, of but a few of the
20 various ways in which the principles of the invention may be employed and the present invention is intended to include all such aspects and their equivalents. Other advantages and novel features of the invention may become apparent from the following detailed description of the invention when considered in conjunction with the drawings.

25 BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 illustrates a general block diagram of aspects of a programming language component, in accordance with the present invention.

FIG. 2 illustrates a flow diagram of the process of the present invention.

FIG. 3 illustrates a protocol exchange diagram between a client and a service
30 provider.

FIG. 4 illustrates a general system implementation of the asynchronous programming language of the present invention.

FIG. 5 illustrates an exemplary computing environment in which the inventions may be implemented.

5 FIG. 6 illustrates a schematic block diagram of a networking environment in accordance with the present invention.

DETAILED DESCRIPTION OF THE INVENTION

10 The present invention is now described with reference to the drawings, wherein like reference numerals are used to refer to like elements throughout. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It may be evident, however, that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form in
15 order to facilitate describing the present invention.

As used in this application, the terms “component” and “system” are intended to refer to a computer-related entity, either hardware, a combination of hardware and software, software, or software in execution. For example, a component may be, but is not limited to being, a process running on a processor, a processor, an object, an
20 executable, a thread of execution, a program, and/or a computer. By way of illustration, both an application running on a server and the server can be a component. One or more components may reside within a process and/or thread of execution and a component may be localized on one computer and/or distributed between two or more computers.

Referring now to FIG. 1, there is illustrated a general block diagram of aspects of
25 an asynchronous programming language (APL) component 100, in accordance with the present invention, as described herein and according to the document in appended hereto, the contents of which are hereby incorporated by reference. There is provided a modular checking component 102 that functions to extract the minimum relative interactive descriptions between a client and a service. In furtherance thereof, the modular checking
30 component 102 includes a model extraction component 104 that extracts a client model and a service (or provider) model from the respective client and service interfaces. The

models are then passed to a model checker component 106 to systematically and methodically determine all of the possible interactions between the two models. Since this is an asynchronous regime, this includes, but is not limited to, delays that could occur when passing information between the client and the service, ordering problems that may occur, error messaging, and the resolution of such potential problems.

Once the model checker 106 determines the possible interactions, the behavioral aspects are well defined such that a “contract” is formed according thereto. The contract is then enforced by a conformance checker component 108 by ensuring that both the client and service conform to the contract. The conformance checker 108 uses the contract to both check that the clients of a respective asynchronous client interface are invoked in the appropriate order and that they handle both normal and exceptional results. The checker 108 also checks that the provider of the asynchronous provider interface is implemented according to its part of the contract. The modular checker 102 is capable of handling the complexity of programs written in a modern, object-oriented programming language. Note that the conformance checker component 108 may receive the results of the model checker component 106 directly therefrom (not shown), instead of through the modular checking component 102.

Referring now to FIG. 2, there is illustrated a flow diagram of the process of the present invention. While, for purposes of simplicity of explanation, the methodology is shown and described as a series of acts, it is to be understood and appreciated that the present invention is not limited by the order of acts, as some acts may, in accordance with the present invention, occur in different orders and/or concurrently with other acts from that shown and described herein. For example, those skilled in the art will understand and appreciate that a methodology could alternatively be represented as a series of interrelated states or events, such as in a state diagram. Moreover, not all illustrated acts may be required to implement a methodology in accordance with the present invention. At 200, the modular checking component utilizes the extraction algorithm to extract models of the both the client(s) and the service(s). The respective models define the minimum interactive communication descriptions for the respective interfaces. At 202, model checking is performed to arrive at all possible interactive scenarios of the given descriptions, and responses thereto. These define the contract terms between the client

and service in the asynchronous exchange. At 204, the contract terms are passed to the conformance checker to ensure that both the client and service adhere to the terms. Flow then reaches a Stop block.

- The disclosed programming language extends the C# programming language with
- 5 constructs for asynchronous programming. The constructs both provide natural abstractions for this type of programming and also facilitate the program analysis described herein. These constructs both provide natural abstractions for this type of programming and also facilitate the program analysis described in this paper. The first is an asynchronous call that returns a future-like handle to the eventual result of the call.
- 10 The second is a synchronization statement similar to the join construct in the join calculus. The third is a facility to state behavioral contracts on service interfaces. The fourth is a transaction directive to delineate scopes of conversations by clients. The fifth is a tracked type qualifier used by the checker. The following Table 1 provides the formal syntax for these language extensions.

15

Table 1. Syntax Extensions for Asynchronous Programming

1. Future		
<i>type</i>	::=	future <i>type</i>
2. Join		
<i>timehead</i>	::=	timeout(<i>int</i>)
<i>param</i>	::=	<i>attributes</i> ^{OPT} <i>parammodifier</i> ^{OPT} <i>type id</i>
<i>fv</i>	::=	<i>Id</i> <i>!id</i>
<i>joinhead</i>	::=	<i>fv</i> ₁ (<i>param</i> ₁ ^{OPT}) &...& <i>fv</i> _{<i>n</i>} (<i>param</i> _{<i>n</i>} ^{OPT})
<i>joinrule</i>	::=	<i>joinhead</i> → <i>stmt</i> <i>timehead</i> → <i>stmt</i>
<i>joinstmt</i>	::=	join{ <i>joinrule</i> ₁ +...+ <i>joinrule</i> _{<i>m</i>} }
3. Contract		
<i>classdecl</i>	::=	<i>classhead classbody</i>
<i>classhead</i>	::=	<i>attributes membermodifiers</i> <i>classident classbase</i> <i>inputcon</i> ^{OPT} <i>corspec</i> ^{OPT}
<i>classbody</i>	::=	{ <i>memberdecl</i> ₁ ... <i>memberdecl</i> _{<i>n</i>} }
<i>memberdecl</i>	::=	<i>methodocon</i> ^{OPT} <i>constructordecl</i> <i>methodocon</i> ^{OPT} <i>methoddecl</i>

		<i>fielddecl</i>
		<i>otherdecl</i>
inputcon	::=	[contract <i>C</i>]
methodcon	::=	[contract <i>P</i>]
<i>C</i>	::=	null $e_1? \rightarrow C_1 + \dots + e_n? \rightarrow C_n$
		$C_1 \parallel \dots \parallel C_n \mid C^*$
<i>P</i>	::=	null $X \mid P \mid P \mid \text{rec } X.P$
		$\text{join } \{I_1 + \dots + I_n\} \mid O_1 \# \dots \# O_n$
<i>I</i>	::=	$e_1? \& \dots \& e_n? \rightarrow P$
<i>O</i>	::=	return throw <i>Exn</i> $ev!; P$
corspec	::=	[contract correlation <i>type</i>]
4. Transaction		
<i>transaction</i>	::=	transaction(<i>svcobjlist</i>){ <i>stmt</i> }
<i>svcobjlist</i>	::=	$\text{var}_1 \text{ corid}_1^{OPT}, \dots, \text{var}_n \text{ corid}_n^{OPT}$
<i>corid</i>	::=	correlate $\text{var}_1 \dots \text{var}_m$
5. Tracked		
<i>type</i>	::=	tracked <i>type</i>

ASYNCHRONOUS CALLS AND FUTURES

In the disclosed programming language, as in Java and C#, procedures are methods in objects. These methods may be called synchronously, in the normal manner, or may be invoked asynchronously. An asynchronously invoked method executes concurrently with its caller. The asynchronous call itself completes immediately by returning a value called a future. The future is a typed handle on the result eventually produced by the method, and it provides a means to explicitly synchronize the two computations.

A future is a first-class value, created at an asynchronous call that can subsequently be stored, passed as an argument, or returned as a result. The futures herein are similar to futures popularized in Multilisp, except for a key difference being that the disclosed programming language requires the result contained in a future be explicitly retrieved through a join statement, rather than allowing it to be implicitly retrieved through a reference to the value of the future. Making this synchronization explicit fits better with the philosophy of a strongly typed language, such as C#, and the join

construct is an expressive notation for describing the synchronization of several computations and for handling failure.

The asynchronous call construct extends C# invocation expressions by prefixing the keyword `async` to an invocation expression, as in:

5

```
    async foo(1);
```

Associated with asynchronous calls are a new type constructor and a new type rule. If `foo` is declared as a method returning a value of type T , then an asynchronous call to `foo` returns a value of type `future T`.

10

The type constructor `future` extends C# type expressions by prefixing the keyword `future` to a type expression. A type of the form `future T` is called a *future type*, and a value of that type is a *T-future*. For example:

15

```
    future int foo(int a) { ... }
    future MyClass bar() { ... }
    void baz(future string F) { ... }
```

20

declares a method `foo` returning an integer future, a method `bar` returning a future object of type `MyClass`, and a method `baz` accepting a string future as argument.

Future variables may be introduced by local declarations:

25

```
    future T MyFuture1 = async foo(2);
    future T MyFuture2;
    MyFuture2 = async bar();
```

or by field member declarations:

```

class MyClass{
    public future int MyFuture3;
5    public void bax(){ MyFuture3 = async bar(); }
    }

```

Referring now to FIG. 3, there is illustrated a protocol exchange diagram 300 between a client 301 and a service provider 303. The client 301 submits an asynchronous call 302 to the service provider 303. The service provider 303 invokes a service, as indicated at 304. At 306, a new future value is returned substantially immediately to its caller, the client 301. The invoked method (or service), at 304, in the service provider 303 executes concurrently with the continuing process of returning the call, at 306. A future value is an object that is in one of three states: *Undefined*, *Normal*, and *Exceptional*. Initially, a future value is passed as *Undefined* to the client 301, as indicated at 306, and the value remains in that state in the client 301 until the invoked service computation of the service provider 303 finishes execution. Note that the order of the invoked service, at 304, and the returned future value, at 306, may be reversed, such that the future value is return immediately, before the service invocation. In either case, both processes are occurring substantially concurrently.

When the service computation completes execution, indicated at 308, a determination is made as to if the execution completed normally (*i.e.*, returns a future value and does not generate an “exception”) or exceptionally. If normally, at 310, the service provider 303 returns a “*Normal*” future value that transitions from the existing “*Undefined*” value in the client 301 to the “*Normal*” state, as indicated at 312. If the service computation does not complete normally, flow is to 314 to determine if the computation completed with an exception. If so, the service provider 303 returns an “*Exceptional*” future value that transitions from the existing “*Undefined*” value in the client 301 to the “*Exceptional*” state, as indicated at 316. A future value makes at most one state transition and then remains in that state. If the service computation does not complete with an exception, flow returns a message indicating such, indicated at 318.

The state of the future value can be examined, and the callee result retrieved, only in a join statement. Any other attempt to examine the state of a future value is a static type error that will be detected by the compiler.

5 JOIN STATEMENTS

A join statement is used to synchronize with computations in a set of futures and to retrieve values from them. The join is a blocking construct, which suspends the execution of a computation until a specified set of futures changes state or a certain interval elapses (*e.g.*, a time out). The programmer specifies the combinations of
10 successful and exceptional future completions that will satisfy the join. For example:

```

    join {
        MyFuture4(T1 v1) & MyFuture5(T2 v2)
        -> Continue(v1,v2);
15      !MyFuture4(Exception e) -> ProcessError(e);
        !MyFuture5(Exception e) -> ProcessError(e);
        timeout(1200) -> throw new TimeoutExn(); }

```

MyFuture4 and MyFuture5 are future values created by asynchronous calls
20 earlier in the program. The statement contains four *join rules*. The first is satisfied if both futures complete normally. In this case, the results of the two calls are bound to the new variables v1 and v2 and the method Continue is invoked. The next two rules handle the cases in which either asynchronous call returns an exceptional result. In either clause, the exceptional value is bound to the new variable e and the method ProcessError is
25 invoked. The final rule specifies a time out interval of 1200 milliseconds for the join clause.

The join statement syntax includes the following. The *future variables* of a rule (*joinrule*) are the identifiers (*id*) referenced in $fv_1 \dots fv_n$ in the *head* of the rule. The *parameters* of a rule are the identifiers in the list $param_{OPT1} \dots param_{OPTn}$. The statements
30 (*stmt*) following a join head are the *continuation statements* of the rule. The syntax class *stmt* is the C# statement and statement-block extended with the join statement *joinstmt*.

The class *id* contains C# identifiers (variable names) and *param* contains C# formal parameters.

The type rules associated with a join statement are: future variables in a join rule must be of future type; in a join rule (*joinhead*), if a future variable not preceded by an exclamation mark fv_i is of type *future T*, then *param* must be declared of type *T*; and, in a join rule (*joinhead*), if a future variable is preceded by an exclamation mark $!fv_i$, then $param_i$ must be declared to be an exceptional type.

The scope of a parameter in a join rule (normal or exceptional) is the continuation statements of the rule, and its occurrence in the head of the rule is binding for that scope.

The rules in a join statement are evaluated in order. The statement executes the continuation of the first rule whose left-hand side is satisfied. A rule is satisfied if all future variables in its head have transitioned out of the *undefined* state and the type of result matches the pattern in the rule. An unnegated future variable (*fv*) matches a future variable in state *normal* and a negated future variable (*!fv*) matches a future variable in state *exceptional*. If the join statement contains a time out rule and no other rule in the join is satisfied in at least the time specified in the time out, then this rule is considered satisfied and it is processed in the normal way.

At most one rule in a join may be satisfied. After the continuation of a satisfied rule completes execution, the join statement terminates and control passes to the subsequent statement in the program.

CONTRACTS

Because asynchronous programming is inherently concurrent, bugs can be subtle and may only manifest themselves in certain interleavings among asynchronous activities. Message reordering in asynchronous message passing makes it very easy to introduce unintended behavior through inadequate synchronization.

The programming language provides a way to statically detect such errors, through source-level *contracts*. Contracts express stateful protocols governing the legal interactions between a client and a service. Contracts extend C# class declarations to express two aspects of the publicly visible behavioral interface of a class: (1) temporal constraints on the allowed sequences of invocations of methods of the class, and (2) the

externally visible behavior of each method. Classes with contracts are referred to as *services*, and the contracts are sometimes called *service contracts*. The contract language is more general than similar efforts in sequential programs, as it models the effects of multiple clients concurrently accessing an interface. The programming language
 5 compiler contains a *conformance* checker, which checks that services and clients obey contracts. The programming language notion of contract conformance and the method for conformance checking are described hereinbelow.

The contract language syntax is as follows. Contracts consist of contract process expressions in C# class declarations (see Table 1). The syntax classes of the contract
 10 extension are given **bold** names. The details of standard C# class declaration syntax are omitted. The syntax class *otherdecl* covers omitted class member declaration forms (constant-, property-, event-, indexer-, operator-, destructor-, and type declarations).

The syntax class *inputcon* denotes contracts for input constraints and the syntax class *methodcon* denotes contracts for method constraints. Input constraints C and
 15 method constraints P are defined in a CSP/CCS-like language. Communicating Sequential Processes (CSP) is a language used to describe and reason about concurrent systems. It consists of a process algebra combined with a functional language. Calculus of Communicating Systems (CCS) is a language that has concurrent composition of processes and synchronous point-to-point communication. Input constraints are
 20 deterministic process expressions that have only input operations and in which parallel composition cannot occur within loops. More precisely, processes C are required to be is deterministic, and $C^* = \text{rec } X.(C[X/\text{null}])$ provided C does not contain \parallel . Contract process expressions that occur in input constraints and in method constraints and may contain *event names* belonging to the syntax class e . These names range over the method
 25 names and exception names that are in scope in the class declaration.

The semantics of input and method constraints is the standard CCS operational semantics extended with assignments and asserts. The character “#” is used to denote internal choice, and “+” to denote external choice.

TRANSACTIONS AND TRACKED TYPES

In order to facilitate conformance checking, the client writer is allowed to delimit the scope of a “conversation,” *i.e.*, an instance of a concurrent interaction between the client and a set of service instances. *Transaction directives* are used within client code to do this. Since multiple simultaneous conversations are possible with a server, the transaction directive supports specifying correlation variables that uniquely identify the associated conversation. Finally, the client writer may single out certain types as being *tracked*, by prefixing the keyword tracked to a C# type expression or class name, as in

```
10      class Conversation{ tracked State MyState;...}
```

The tracked modifier informs the programming language conformance checker that the `State` field of the `Conversation` class is relevant for checking the client against a contract within a transaction scope. The use of transactions and tracked types in conformance checking is explained in further detail hereinbelow.

EXAMPLES

The following Example 1 code contains the interface for a simple service that models an airline reservation system.

```
20
```

Example 1. Sample Airline Reservation System

```

[WebService]
public class Airline
25  [contract (Query? -> ( Buy? -> Cancel? + Cancel? ))* ]
    [contract correlation Cid]
    {
        [contract {throw AirlineException # return} ]
        future Info Query(Cid PO,string itinerary);
30    [contract {throw AirlineException # return} ]
        future Info Buy(Cid PO);
        [contract {throw AirlineException # return} ]

```

```

        future Info Cancel(Cid PO);
    }

```

The code exposes three entry points: Query, Buy and Cancel. A client must first call Query to check availability of a seat. Then it can either buy or cancel the reservation by calling Buy or Cancel, respectively. Furthermore, after purchasing the seat, the client of the airline is still allowed to cancel. Once canceled, the client can possibly start another query. The input constraint (Query? -> (Buy? -> Cancel? + Cancel?)) * succinctly prescribes these sequencing constraints. The input constraint only specifies the set of all allowable sequences of method invocations, and does not force the client to actually make any method calls. For example, a client that calls Query followed by Buy is not necessarily forced to call Cancel. In addition, the method constraint throw AirlineException # return specifies that the body of the Query method can either return normally or throw an AirlineException.

As an example of using this service, suppose it is desired to build a travel agent to coordinate a reservation with an airline service (*Airline*) and a car-rental service (*Rental*) with the following objective: either purchase both the airline and car reservations, or cancel both. Assume that the service *Rental* has an interface isomorphic to the *Airline* service from Example 1. The following Example 2 illustrates a sample travel agent client.

Example 2. Sample Airline-Rental Client

```

Class TravelAgent{
    private Airline myAir;
    private Rental myRental;
    private CId myPO;
    void doAirlineAndRentalREservation() {
        myAir = new Airline();
        myRental = new Rental();
        myPO = new Cid();
        transaction(myAir correlate myPO,

```

```

        myCar correlate myPO) {
        String itinerary = User.getItinerary();
        Future Info fAir =
            async myAir.Query(myPO, itinerary);
5         Future Info fRental =
            async myRental.Query(myPO, itinerary);
        join { fAir & fRental      -> buyBoth();
            + !fAir      -> cancelBoth();
            + !fRental   -> cancelBoth();
10        }
        }
    }
}

```

The transaction directive is used to delimit the scope and specify correlation
 15 variables for the conversation. In this case, myPO is used as correlation for both the
 services. Note how the join construct is used to wait for both the futures fAir and
 fRental to complete.

The following Example 3 shows a buggy client that will be used as an example
 herein.

20

Example 3. Buggy Client Code

```

Class SimpleClient{
private Airline myAir;
25 private Cid myPO;
void startReservation() {
    myAir = new Airline();
    myPO = new Cid();
    //now interact with myAir using
    //myPO as the correlation id
30    transaction (myAir correlate myPO) {
        String itinerary = User.getItinerary();
        Future Info fInfo =

```

```

        async myAir.Query(myPO, itinerary);
        async myAir.Buy(myPO);
    }
}

```

5

Although the client calls `Query` before calling `Buy`, the calls are asynchronous, so `Buy` can potentially be invoked before the call to `Query` completes. A correct client should synchronize on `fInfo` before calling `Buy`.

10 CONFORMANCE CHECKING

The disclosed programming language compiler checks if components of the program *conform* with the contracts that are specified on the interfaces. The checker separately checks for two kinds of conformance: (1) *client conformance*, which means that a *client* (a program using a service interface with contracts) obeys the contract of the service it uses, and (2) *service conformance*, which means that a *service* (an implementation of an interface with contracts) obeys the contract it implements. Client and service conformance are technically distinct forms of correctness. Intuitively, client conformance requires that the client code not violate the input constraints of contracts and handle all exceptions that the server can potentially raise. Service conformance requires that the service code implementing each service method is a refinement of its method constraint in the contract.

The fact that both notions of conformance only reference the contract to represent the other party (client or service) is the key to *modular* conformance checking. The provider of a service need not expose its implementation to clients. Similarly, the service can be written and evolved independently of its clients. Moreover, checking service conformance one method at a time further modularizes the problem of conformance checking. Because conformance checking relies on model checking, as described below, modularity is essential to mitigate the state explosion in conformance checking.

STATE

The contracts of the disclosed programming language are stateful. For example, the input constraint shown in Example 1 is initially in a state where only a call to `Query` is accepted. It then transitions to another state where either a call to `Buy` or to `Cancel` is accepted, and so forth. In addition, the state of the contract after a call to `Query` also indicates that the service may respond according to the method constraints on the `Query` method. The state of a contract is referred to as *contract state*.

Clients and services also have states, in their respective implementations. For example, the client shown in Example 2 enters states where two asynchronous calls are made. It then enters a state where it is ready to join on the services' responses. The state of a client implementation is referred to as *client state*, and the state of a service implementation, as *service state*. The essence of conformance checking is to relate these three kinds of states (*i.e.*, the contract state, client state, and service state). Since clients run concurrently with the services they use, checking client conformance involves exploring the concurrent composition of the client state machine with the contract state machines of the services. Checking service conformance requires a comparison of the service state machine with the contract state machine according to a refinement relation.

MODELS

The programming language conformance checker relies on model checking to explore the relation between contract state, client state and service state. Using a type-based technique that is described in greater detail hereinbelow the checker extracts a *model* from each component of the program being analyzed. A model is an abstraction of a program component containing only communication actions, and operations that are *relevant* for maintaining the state of the communication protocol. Models are expressed in a model intermediate language, **mIL**, presented in the end of this section.

Conformance checking is done in two steps. First, various **mIL** models—called *interface models*, *client models*, and *service models* are constructed from the disclosed program. Next, these models are fed into a model checker to check for client conformance and service conformance.

An *interface model* is constructed from a contract specification. It consists of a set of **mIL** process definitions, one for each method of the interface. Construction of interface models from contracts is described hereinbelow.

Client models and *service models* are constructed by extraction from the programming language code for the client and service respectively. Because client and service models need to represent the relevant communication actions of implementation code, models must soundly represent the objects on which those actions occur. Relevant data is those data structures that are directly contract relevant (such as futures, correlation identifiers), and that therefore need to be represented in models. In addition, the programmer can also indicate certain program variables as relevant using the tracked modifier. A key problem in model extraction is *abstraction*: to soundly represent relevant data and operations on them, while throwing away as much as possible of the program that is irrelevant to conformance. Since the programming language error detection is sound, pointer aliasing is a concern. For example, if x is relevant, the statement $y.foo()$, may not be soundly ignores, if y and x might be aliased.

A novel feature of the model extraction algorithm is the use of *regions* to soundly model relevant data. To illustrate, the interface model for Example 1, the Airline service, is compiled into three **mIL** processes: E_{Query} , E_{Buy} , and E_{Cancel} . An asynchronous call in a client is then represented by,

```
Future Info fAir = async myAir.Buy(myPO, itinerary)
```

by the **mIL** model statements,

```
LOCAL fAir = NEW[ $\tau_F$ ] CHAN; RUN  $E_{Buy}$  [ $\tau_{myAir}$ , fAir]
```

Here, the future $fAir$ is represented as a new **mIL** channel, and the asynchronous call is modeled by spawning an instance of the interface process E_{Buy} . The names τ_{myAir} and τ_F refer to regions. Model extraction is described hereinbelow.

mIL and mILCruncher

The model checker, called herein mILCruncher, operates on models expressed in the **mIL** language. The following Table 2 contains the syntax for **mIL**.

5 Table 2. mIL Syntax.

e	$::=$	true false $*$ x CHOOSE $[r]$ NEW $[r]$ T $a[\vec{r}](\vec{e})$ ef
S	$::=$	$e!k$ $e?k$ $a[\vec{r}](\vec{e})$ RUN $a[\vec{r}](\vec{e})$ ATOMIC $\{S\}$ JOIN $\{j_1 + \dots + j_n\}$ DELETE $[r]$ $e_1 = e_2$ IF $(e)\{S_1\}\text{ELSE}\{S_2\}$ TRY $\{S\}\text{CATCH}(Exn)\{S\}$ THROW Exn LET $x = e$ IN e GOTO l $l:S$ $S;S$
j	$::=$	$x_1 \& \dots \& \rightarrow S$
D	$::=$	$a[\vec{r}](\vec{x})\{S\}$ REF $C \{T_1 f_1, \dots, T_n f_n\}$ ENUM $E \{k_1, \dots, k_n\}$ EXCEPTION Exn LOCAL T x
T	$::=$	BOOL CHAN SET C E Exn

Beyond usual types and control flow constructs, **mIL** provides features useful in modeling software. Futures are modeled with channels for send ($e!k$) and receive ($e?k$), and possibly asynchronous message queues used for inter-process communication.

- 10 Receive operations can be collected together in a join construct, similar to that in the programming language. The **mIL** language provides conventional, synchronous function calls and a **RUN** construct, which can model the programming language's asynchronous function call. Statement blocks in **mIL** can be enclosed in an **ATOMIC** modifier, which is a model checker directive to execute the block atomically, thereby reducing the state
- 15 explosion that arises from interleaving the statements in the block with other concurrent activities. In addition to usual value types, **mIL** has a notion of a *region*, which is a heap-allocated set of objects. There are two constructs used to model regions: **NEW** and **CHOOSE**. The expression **NEW** $[r]$ T allocates a new object of type T in region r . **NEW** $[]$ **SET** creates a new region that is an empty set, and **CHOOSE** $[r]$ chooses a
- 20 member of region r nondeterministically. **DELETE** $[r]$ deletes a region, when no concurrent processes can reference it.

The scope of every client conversation is clearly marked in a program of the present invention using transaction directives. Every transaction directive in the program results in a client conformance check. To check for client conformance, the model

25 extracted from client code and the interface models for the services listed in the

transaction directive are fed into mILCruncher. mILCruncher then explores all possible interleavings of the asynchronous activities to check if the interface contract can be violated in any such interleaving. For example, when applying the client conformance check on the client in Example 3, mILCruncher is able to detect an interleaving where the effect process for the Buy method gets scheduled before the effect process for the Query method leading to an assertion violation in the effect process for the Buy method.

Service conformance is checked one service method at a time. For each method a , a model B_a is extracted from the body of the method, and checked for conformance against method a 's method specification S_a . More precisely, B_a and S_a are mILCruncher models and mILCruncher checks if $B_a \leq S_a$, where “ \leq ” is the conformance relation. Our current implementation of mILCruncher uses the SPIN model checker as a back-end. SPIN is a product of Lucent Technologies, Inc., and is a software verification tool that checks the logical consistency of a specification. Both client conformance and service conformance are translated into invariant checking queries on SPIN.

INTERFACE MODELS

An interface model consists of a set of **mIL** process definitions \mathbf{E}_a , called *effect processes*. There exists one effect process for each method a of the interface. Intuitively, the process \mathbf{E}_a models the method a with respect to the effects that a call to a has on the contract state, the future created at an asynchronous call to a , and the futures passed into a .

An effect process \mathbf{E}_a has two components: a process C_a that models the effect of the call a on the contract state that is implicitly specified by the input constraint C of the contract, and a process $mil(P)$ that models the method constraint P of the contract on a . Process $mil(P)$ can be constructed from the method constraint P in a straightforward way. More precisely, the effect process \mathbf{E}_a is parameterized by two names, r_S and c_F . The first parameter, r_S , is a region name used to represent the state of the contract. The second parameter, c_F , is an **mIL** channel parameter representing the future created at an asynchronous call. The effect process \mathbf{E}_a is defined by,

$$\mathbf{E}_a[r_S, c_F] \{ \text{ATOMIC} \{ \sigma_{r_S, c_F}(C_a) \}; \sigma_{r_S, c_F}(mil(P)) \},$$

where σ_{r_S, c_F} is a map that translates return and throw statements into message passing operations on the channel c_F and state variable accesses into references of the state region name r_S , (1) for each state variable x_i : $\sigma_{r_S, c_F}(x_i) = r_S.x_i$, (2) for each return statement in P : $\sigma_{r_S, c_F}(\text{return}) = c_F! \text{RETURN}$, (3) for each throw statement in P : $\sigma_{r_S, c_F}(\text{throw } \text{Exn}) = c_F! \text{Exn}$.

The parameterization of E_a on state r_F and c_F is essential for *instantiating* effect processes in the models: different calls into a service belonging to the same contract instance (transaction) can affect the same contract state in **mIL** models, by having the model pass the same state variable r_S at each call. Calls belonging to different instances pass different state variables. Calls create different futures, on which responses come back from the service, which is modeled by passing different channels c_F to effect processes.

The algorithm for constructing C_a is as follows. The input constraint C is considered as a rooted labeled transition graph, $\mathbf{G}(C)$ with root $\mathbf{r}(C)$. The nodes of $\mathbf{G}(C)$ are labeled by `null`, `->`, and `||`. The arcs are either unlabeled or labeled by the method names a_i occurring in C . The following four rules translate C into a labeled transition graph by induction on C : (1) If $C = \text{null}$, then $\mathbf{G}(C)$ is a single node $\mathbf{r}(C)$ labeled `null`; (2) If $C = a_1? -> C_1 + \dots + a_n? -> C_n$, then $\mathbf{r}(C)$ is a new node labeled `+`, with n new child nodes, each labeled `->`. The arcs from $\mathbf{r}(C)$ to the child nodes are unlabeled. For each i^{th} child node, there is an arc labeled a_i from the child node to $\mathbf{r}(C_i)$; (3) If $C = C_1 || C_2$, then $\mathbf{r}(C)$ is a new node labeled `||` with two child nodes, $\mathbf{r}(C_1)$ and $\mathbf{r}(C_2)$. The arcs from $\mathbf{r}(C)$ to $\mathbf{r}(C_1)$ and $\mathbf{r}(C_2)$ are unlabeled.; (4) If $C = C_1 * = \text{rec}X.(C_1[X/\text{null}])$, then $\mathbf{r}(C) = \mathbf{r}(C_1)$, and $\mathbf{G}(C)$ arises from $\mathbf{G}(C_1)$ by adding unlabeled edges from the nodes labeled `null` back to $\mathbf{r}(C)$.

Note that due to restrictions on the syntax, a node labeled with `||` cannot be part of a cycle in $\mathbf{G}(C)$. Using $\mathbf{G}(C)$, a set of state machines are assigned to C as follows. A *sequential subgraph* of $\mathbf{G}(C)$ is a maximal subgraph of $\mathbf{G}(C)$, whose root node is either $\mathbf{r}(C)$ or a node whose parent in $\mathbf{G}(C)$ is labeled `||`, and such that it has nodes labeled `||` only at leaf nodes (nodes with no arcs leaving them). Each sequential subgraph represents a sequential state machine. Each j^{th} machine is encoded using a state enumeration type

$\{n_{j_1}, \dots, n_{j_m}\}$ naming the j_m individual nodes in the subgraph, together with an enabled bit b_j indicating whether the machine is active. In order to generate the method constraints for method a , let $\langle n, a, n' \rangle$ denote an a labeled edge from node n to node n' in $G(C)$. Let $M_{\langle n, a, n' \rangle}$ be the sequential state machine in which the a -labeled edge from n to n' occurs (by determinism of C , there is exactly one such machine), and let $\langle n_1, a, n'_1 \rangle, \dots, \langle n_k, a, n'_k \rangle$ be all the a -labeled edges in $G(C)$. Then process C_a becomes,

```

10  if( $M_{\langle n_1, a, n'_1 \rangle}.b == \text{true} \ \&\& \ M_{\langle n_1, a, n'_1 \rangle}.state == n_1$ )
      then {  $M_{\langle n_1, a, n'_1 \rangle}.state = n'_1; S_1$  }
      ...
      else if( $M_{\langle n_k, a, n'_k \rangle}.b == \text{true} \ \&\& \ M_{\langle n_k, a, n'_k \rangle}.state == n_k$ )
          then {  $M_{\langle n_k, a, n'_k \rangle}.state = n'_k; S_k$  }
          else assert(false);

```

15 where $M_{\langle n_1, a, n'_1 \rangle}.b$ denotes a state variable holding the enabled bit, and $M_{\langle n_1, a, n'_1 \rangle}.state$ is a state variable ranging over the node states of the machine $M_{\langle n, a, n' \rangle}$. The statements S_i in the then-statements above are defined to be empty, if the node n'_i is not labeled with \parallel . But if it is, then S_i is the statement $M_{i_1}.b = \text{true}; \dots, M_{i_q}.b = \text{true};$, where the $M_{i_1} \dots M_{i_q}$ are the state machines associated with the components of the parallel composition at node n'_i .

20 For example, consider the input constraint C from Example 1. The graph $G(C)$ has three nodes n_1, n_2 , and n_3 , labeled with \rightarrow , $+$ and \rightarrow respectively with n_1 as the root node. The graph has four edges $\langle n_1, \text{Query}, n_2 \rangle, \langle n_2, \text{Buy}, n_3 \rangle, \langle n_3, \text{Cancel}, n_1 \rangle, \langle n_2, \text{Cancel}, n_1 \rangle$. The state variable for this sequential machine has the enumerated type $\{n_1, n_2, n_3\}$. The effect process $E_{\text{Query}}[r, c]$ is given by:

```

25   $E_{\text{Query}}[r, c]\{$ 
      ATOMIC{IF ( $r.M.b == \text{true} \ \&\& \ r.M.state == n_1$ )
          THEN  $r.M.state = n_2$ ; ELSE ASSERT(false);}
      IF (*) THEN  $c!\text{AirlineException}$ ; ELSE  $c!\text{RETURN}$ ;
30  }

```

PROGRAM MODELS

The extraction algorithm uses a new combination of region type systems and model checking. Objects that are *relevant*, i.e., that need to be represented in the model,

are associated with *region names* in an underlying type system. Regions represent equivalence classes of objects based on aliasing relationships in the program. These equivalence classes are represented by region names produced as a result of type inference. The region type system guarantees that two variables with different region names can never point to the same object. Two variables with the same region name may (or may not) point to the same object.

To be sound, the **mIL** models extracted must contain all changes to objects that are *relevant*. This is achieved by explicitly representing the association between relevant objects and their containing regions in the **mIL** model. There are several consequences of such a representation. An assignment such as $y.val = 10$ is abstracted differently depending on whether y is relevant or not. If y is not relevant, then the assignment will be omitted from the model. This is sound because the region type system guarantees that y will never alias any relevant object. If y is relevant and is associated with a region name r , then the assignment is represented in the **mIL** model as $LET\ t = CHOOSE[r]\ IN\ t.val = 10$, resulting in the model checker nondeterministically choosing an object in the region r and updating its `val` field to ten.

A test such as $if\ (x.val < 5)$ in the program of the present invention is also abstracted differently depending on whether x is relevant. If x is not relevant, the model contains a nondeterministic choice $IF\ (*)$. If x is relevant and is associated with region p , then the test is abstracted as $LET\ t = CHOOSE[p]\ IN\ IF(t.val < 5)$, resulting in the model checker non-deterministically choosing an object in the region p and inspecting the value field of the object. An assignment such as $x = y$ where x and y are references to an object potentially containing relevant fields does not need to be represented in the **mIL** model, since the region type system ensures that x and y have the same region name.

The region type system is polymorphic to increase precision, which results in polymorphic region variables appearing as parameters to methods. Consequently, the **mIL** model contains region variables, which are instantiated with region names during method calls inside the model checker. For simplicity, primitive types such as integers are “boxed”, so they can be tracked by region annotations.

In general, relevant objects associated with protocols will not be heavily aliased. In the examples considered, regions associated with relevant objects tend to be singleton sets containing just one object, and consequently no loss of precision or efficiency occurs in the model checker. However, the extraction scheme guarantees soundness even in the presence of aliasing, at the cost of forcing the model checker to explore more paths and reduce the precision. At each object access, the model checker will have to explore a number of nondeterministic choices proportional to the size of the regions.

REGIONS AND RELEVANT TYPES

Following conventional practices, region types have the form (τ, p) , where a type τ is paired with a *place* p . A place is either a *region name*, τ , or a *region variable*, p . Region variables represent region parameters on methods, and region names represent actual regions. In the instant system, assigning a region type (τ, p) to an object marks it as *relevant*: in addition to having type τ , a relevant object belongs to a unique region p , which will be represented in the program model. Futures, service objects, and correlation identifiers are automatically regarded as relevant. *Tracked* types, as specified by the programmer with `tracked` declaration, are automatically relevant. Non-relevant data are assigned non-relevant types of the form τ without a region name. Hence the disclosed region system is partial, in that only relevant data are directly assigned region types. This differs from conventional methods. The type system ensures that relevant types are assigned consistently, so that relevant values may only be assigned to relevant variables and passed to methods with relevant parameters.

Region information is illustrated in the continuing examples. The following Example 5 shows the Airline client with region information.

Example 5. Client with Region Information

```
class SimpleClient< $\rho'_0, \rho'_1$ > {
  private Airline myAir: (AIRLINE,  $\rho'_0$ );
  private Cid MyPO: (CID,  $\rho'_1$ );

  void startReservation[ $\rho_0, \rho_1$ ] (SimpleClient[ $\rho_0, \rho_1$ ] this)
  {
```

```

myAir AT  $\rho_0$  = new Airline() AT  $\rho_0$ ;
MyPO AT  $\rho_1$  = new Cid() AT  $\rho_1$ ;

//now interact with myAir using
//MyPO as the correlation id
transaction (myAir correlate MyPO) {
  String itinerary = User.getItinerary();

  LETREGION  $\rho_{c_1}$  {
    Future Info info1 AT  $\rho_{c_1}$  =
      async Query[ $\rho_0, \rho_1$ ] (myAir, MyPO, itinerary);

    LETREGION  $\rho_{c_2}$  {
      Future Info info2 AT  $\rho_{c_2}$  =
        async Buy[ $\rho_0, \rho_1$ ] (myAir, MyPO); } }
  } }

```

Example 5 shows the result of region inference over the relevant Airline client from Example 3. This information guides the abstraction into **mIL**. However, at this stage, it simply makes explicit the type structure of the region-analyzed program.

- 5 Variables `myAir` and `MyPO` are assigned relevant types of the form (τ, ρ) , because `myAir` is specified as a service object, and `myPO` as a correlation identifier in the `transaction` construct. The underlying types of variables of relevant types are explicitly expressed, as in `myAir: (AIRLINE, ρ'_0)`. When objects of relevant type are dereferenced, the region is mentioned, as in `new Airline() AT ρ_0` (it has done so only in some cases, for brevity).

- 10 On method declarations, the parameters of the region are written over which the method's polymorphic region type is quantified, as in `startReservation [ρ_0, ρ_1](...)`. Methods are assigned region-polymorphic types of the form $\forall \alpha \rho \varepsilon (\mu_1, \dots, \mu_n) \rightarrow {}^\varphi \mu$, with α ranging over types, ρ over region variables, and ε over effects. The region variables mention the regions the method touches. In addition, φ is a static effect set, which
- 15 indicates which actions the method performs on the regions it touches.

In the example, the method `startReservation` operates on relevant data stored in the fields `myAir` and `MyPO`. Region inference therefore infers that

startReservation needs to be passed the corresponding region variables as parameters. At call sites, region names are passed explicitly, as in `Query[ρ_0, ρ_1](...)`.

Region type parameters are made explicit in class types, as in `SimpleClient[ρ_0, ρ_1]` (references to the `this` parameter were elided in the method, for brevity). The parameters range over the region variables of the relevant types (transitively) contained in the class. This is essential for tracking operations on regions that are made indirectly through non-relevant data. In the example, the `SimpleClient` class itself is not relevant, but its fields are. To track operations on the fields of a `SimpleClient` object, the regions must be represented in the region parameters of the class type. The annotation `LETREGION $\rho\{S\}$` introduces a new region with static name ρ in the scope S .

EXTRACTION ALGORITHM

The extraction algorithm is formulated on a core sub-language of the novel programming language with region information, as illustrated in Table 3.

Table 3. Core Language With Region Information.

e	$::=$	$c x \text{ new } C() \mid \text{foo}[\vec{p}] (\vec{x}) \mid e.f \mid e \text{ AT } p$
S	$::=$	$e = e' \mid \text{local } Tx \text{ AT } p \mid \text{if } (e) \{S_1\} \text{ else } \{S_2\}$ $\mid S_1; S_2 \mid \text{LETREGION } \rho \{S\}$ $\mid x \text{ AT } p = \text{async foo}[\vec{p}] (\vec{x}) \{S\} \mid \text{join } (j_1, \dots, j_n)$ $\mid \text{transaction}(o \text{ AT } p_o \text{ correlate } x \text{ AT } p_x) \{S\}$
j	$::=$	$x_1 \text{ AT } p(T_1 y_1) \ \&\dots\& \ x_m \text{ AT } p(T_m y_m) \rightarrow \{S\}$
D	$::=$	$\text{foo}[\vec{p}] (\mu_1 x_1, \dots, \mu_n x_n) \{S\}$ $\text{class } C\langle \vec{p} \rangle \{ \mu_1 f_1; \dots \mu_n f_n; \}$

The following Table 4 presents the algorithm as a translation from core programming language to **mIL**. What is shown is how relevant objects are modeled, using region information.

Table 4. Translation of Programming Language into **mIL**

Expressions (e)	
1	$\llbracket \text{foo}[\bar{p}] (\bar{x}) \rrbracket = \text{foo}[\bar{p}]$
2	$\llbracket \text{new } C() \text{ AT } p \rrbracket = \text{NEW}[p] \llbracket C \rrbracket$
3	$\llbracket e \text{ AT } p \rrbracket = \text{CHOOSE}[p] \quad (\text{where } e \text{ is flattened})$
Statements (S)	
4	$\llbracket e = e' \rrbracket = \llbracket e \rrbracket = \llbracket e' \rrbracket$
5	$\llbracket \text{if}(e) \{S_1\} \text{else} \{S_2\} \rrbracket = \text{IF}(\llbracket e \rrbracket) \{ \llbracket S_1 \rrbracket \} \text{ELSE} \{ \llbracket S_2 \rrbracket \}$
6	$\llbracket \text{local } T \text{ } x \text{ at } p \rrbracket = \text{LOCAL } \llbracket T \rrbracket \text{ } x$
7	$\llbracket S_1 ; S_2 \rrbracket = \llbracket \text{Red}(S_1) \rrbracket, \llbracket \text{Red}(S_2) \rrbracket$
8	$\llbracket \text{LETREGION } \rho \{S\} \rrbracket = \text{LOCAL SET } r = \text{NEW}[] \text{ SET};$ $\llbracket S[r/\rho] \rrbracket;$ $\text{DELETE } [r]$
9	$\llbracket x \text{ AT } p = \text{async foo}[p_0, \bar{p}] (x_0, \bar{x}) \{S\} \rrbracket =$ $\text{LOCAL CHAN } c_F = \text{NEW}[p] \text{ CHAN};$ $x = c_F;$ $\text{RUN } E_{\text{foo}}[p_0, c_F];$ $\llbracket S \rrbracket$
10	$\llbracket \text{join}\{j_1, \dots, j_n\} \rrbracket = \text{JOIN}(\{ \llbracket j_1 \rrbracket \} + \dots + \{ \llbracket j_n \rrbracket \})$
11	$\llbracket x_1 \text{ AT } p(T_1 y_1) \ \& \dots \& \ x_m \text{ AT } p(T_m y_m) \rightarrow \{S\} \rrbracket =$ $x_1 \ \& \dots \& \ x_m \rightarrow \{ \llbracket S \rrbracket \}$
12	$\llbracket \text{transaction}(o \text{ AT } p_o$ $\text{correlate } x \text{ AT } p_x) \{S\} \rrbracket = \llbracket S \rrbracket \{ p_o \mapsto p_x \}$

The behavior of a program is projected with region information to the relevant regions.

- 5 (1) Operations that have *static effects* (in terms of the underlying region-effect type system) on relevant regions are modeled. In particular, allocations into relevant regions are executed by having the model checker allocate objects in the region; dereferencing an object field in a relevant region is executed by having the model checker

non-deterministically choose an object in the region and dereferencing it; methods are modeled as functions of their region parameters.

(2) Statements that have no static effects on relevant regions are omitted from the model.

5 (3) Control decisions based on the values of non-relevant objects are interpreted by the model checker as non-deterministic choice.

Table 4 contains the key features of the disclosed extraction algorithm. The core language considers a method as a function of the object to which it belongs, which is passed as its first parameter (*this*-parameter). Hence, all symbolic memory references are through local variables or parameters, represented by x in the core language. Equation 3 of Table 4 assumes that expressions e are flattened, so that all intermediate expressions are explicitly named through locals. For example, $y = e.f$ is written as $\text{tmp} = e; y = \text{tmp}.f$. The extraction function in Table 4 omits optimizations, which are described later. For brevity, region information $\text{AT } \rho$ is left out in the examples below.

15 The model of a method (Equation 1 of Table 4) is a function of its region parameters. A method call $f_{\circ\circ} [\vec{p}] (\vec{x})$ becomes $f_{\circ\circ} [\vec{p}]$ in the model—actual parameters are projected away and region names kept. This soundly models methods that access relevant data through non-relevant parameters, *without explicitly modeling non-relevant objects*. For example, let C be a non-relevant class with relevant field state, and suppose `bar` is defined by:

```
void bar[ $\rho$ ]( $C[\rho]$  x){ x.state = new State() AT  $\rho$ }
```

(region information is omitted on `x.state` for brevity). At a call, `bar [ρ_0] (x0)`, the region ρ_0 is passed to the method model: `bar[ρ] {NEW[ρ][State]}`. The assignment does not need to be modeled, as explained hereinbelow. Since region inference only constructs *local* region equivalence classes, parameter-aliasing created by calls is modeled by region passing in the model. For example, consider the `baz`:

```
30 void baz [ $\rho_0, \rho_1$ ] ( $C[\rho_0]$  x,  $C[\rho_1]$  y) {
    x.state = new State() AT  $\rho_0$ ;
```

```
y.state = new State() AT  $\rho_1$ 
```

At a call, $\text{baz} [p_0, p_0] (x_0, x_0)$, the region ρ_0 associated is passed with x_0 to the method model, so the call becomes $\text{baz} [p_0, p_0]$. The model of baz is:

5

```
void baz [ $\rho_0, \rho_1$ ] {NEW [ $\rho_0$ ] [State]; NEW [ $\rho_1$ ] [State]}
```

and hence, the call in the model captures that there are two allocations into memory region p_0 . Not all calls to methods need to be modeled. If foo is:

10

```
void foo (C [ $\rho$ ] x, C [ $\rho$ ] y) { x.state = y.state }
```

call need not be modeled to foo , since the type rules already identified that the region parameters associated with x and y are in single region, ρ , and the effect of the

15

assignment is taken into account that way. Correctly distinguishing such cases is done symmetrically with the effect types of the region type system.

Allocation (Equation 2 of Table 4) $\text{new } C() \text{ AT } p$, into region p becomes $\text{NEW}[p][C]$, which creates a new model object in region p , with **mIL** format $[C]$. The format $[C]$ of the **mIL** object contains only relevant fields from C . If C is a *service class*, then it is treated specially, so that $[C]$ denotes the *contract state structure* associated with C 's contract.

20

Memory references (Equation 3 of Table 4) $e \text{ AT } p$ are modeled uniformly in Table 4 by non-deterministically choosing an object in region p , by $\text{CHOOSE}[p]$. However, it is observed (omitting details for lack of space) that symbolic memory references should be treated through locals and parameters differently. References through a local can be modeled concretely by a local in the model, but when referencing relevant data through a parameter should fall back on the region abstraction, so a parameter reference $x_{prm} \text{ AT } p$ becomes $\text{CHOOSE}[p]$. This abstraction is still needed, since relevant objects are passed-by-region in the model. For example, if x_{prm} is a

parameter, then $\text{local } y \text{ AT } p = x_{prm} \text{ AT } p$ becomes modeled in **mIL** by $\text{LOCAL } y = \text{CHOOSE}[p]$.

30

Statements are modeled component-wise (Equations 4-12 of Table 4). Locals of relevant type are represented in the model as **mIL** locals (Equation 6). For the LETREGION statement (Equation 8), the first **mIL** statement, $\text{LOCAL SET } r = \text{NEW}[]$

SET, creates a new region (SET) and binds it to r . The second translates the scope S of the LETREGION, while binding the region variable ρ to the newly created region. Finally, the third statement, DELETE[r], deallocates the region r at the end of the scope, *provided* no live processes reference r (reference-counting is used).

5 An essential part of model extraction consists in model *reduction*, that omits statements unnecessary either because they do not affect or depend on relevant data, or because region types subsume their effects on relevant data. Model reduction is captured by function *Red* in Equation 7 of Table 4. Its definition relies on *static effects*, which are part of conventional region type systems. Every source statement is assigned a set of
10 static effects, of the form $\text{read}(p)$, $\text{write}(p)$, $\text{new}(p)$, $\text{comm}(p)$, which describe, in abstract form, the effect, if any, of the statement on relevant regions. To illustrate, writing $\text{Eff}(S)$ for the static effect of statement S , results $\text{Eff}(\text{new } C() \text{ AT } p) = \{\text{new}(p)\}$, $\text{Eff}(x \text{ AT } p) = \{\text{read}(p)\}$, $\text{Eff}(x \text{ AT } p = c) = \{\text{write}(p)\}$, $\text{Eff}(\text{join } x \text{ AT } p \rightarrow \{S\}) = \{\text{comm}(p)\}$. Statements with no side-effect on relevant regions are assigned empty effect
15 sets and can be elided from the model. Writing “_” for “no region,” for example, results in $\text{Eff}(\text{new } C() \text{ AT } _) = \emptyset$, $\text{Eff}(x \text{ AT } _) = \emptyset$. The effect analysis must be fine-grained, as some statements have effects in some parts but not others. For example, in $x \text{ AT } _ = \text{foo}[p](y)$ the call to foo must be modeled if foo has an effect on relevant region p , even though the assignment to x can be omitted. In general, assignments (other than
20 assignments to relevant locals are kept) can be elided, because region name unification by the type rules takes them into account. For example, $(x.\text{state}) \text{ AT } p = y.\text{state} \text{ AT } p$ can be elided, because the region name p is associated with both references.

The model of an asynchronous call (Equation 9 of Table 4) must represent two things: (1) the *future* value created at the call, and (2) the call’s effect on the interface
25 contract of the called method. Consider the call $x \text{ at } p = \text{async } \text{foo}[p_0, \bar{p}](x_0, \bar{x})\{S\}$, where S is the continuation of the call. The future created at this call is modeled by a new **mIL** channel, C_F , which is allocated into the relevant region p associated with the return value of the call. After assigning the future to the local x , the model executes the **mIL** statement $\text{RUN } \mathbf{E}_{\text{foo}}[p_0, C_F]$ that spawns an instance of the effect process \mathbf{E}_{foo} .

30 In the interface process, call $\mathbf{E}_{\text{foo}}[p_0, C_F]$ passes the region name p_0 associated with the service object containing foo (*this* parameter, x_0) and the future channel, C_F , to the

effect process. Recall that the region parameter for a service object (p_0 in this case) holds the *service contract state*. Passing the region threads contract state through clients of a contract, so that the model correctly maintains contract state across multiple calls to the same interface instance. A call to f_{oo} through a different service object affects an

5 independent instance of the contract state.

A future channel, c_F , is also passed to the interface process. Recalling previously, that the method constraints of f_{oo} are compiled into communication actions on the channel parameter E_{foo} . This correctly threads the contract's communication effects across distinct asynchronous calls. For example, two calls to f_{oo} (even through the same
10 service object) give rise to distinct communication events in the **mIL** model, as they operate on distinct channels. However, these calls correctly affect the same contract state.

A transition directive enclosing a statement S is translated to the **mIL** statement $[S]\{p_o \mapsto p_x\}$, where p_o is the region of the service object o , and p_x is the region of the
15 correlation identifier x . A transaction scope identifies a particular contract instance, so that, within S , only asynchronous calls that pass a correlation identifier within a region p_x to service object p_o will be modeled in that scope, and hence the translation $[S]\{p_o \mapsto p_x\}$ is qualified by the association of p_x to p_o . If no correlation is specified, then all calls addressed to the service object p_o are modeled. Transactions may occur in loops or
20 triggered by external events unbounded in number. In such cases, the transaction directive scopes conformance checking to one generic instance of a conversation.

Applying the algorithm to the client method in Example 5 produces the model in the following Example 6.

25 Example 6. Model of Airline Client.

```

StartReservation[ $\rho_0, \rho_1$ ] {
  NEW[ $\rho_0$ ][Airline];
  NEW[ $\rho_1$ ][Cid];
  LOCAL  $r_{c_1}$  = NEW[] SET;
  LOCAL info1 = NEW[ $r_{c_1}$ ] CHAN;
  RUN  $E_{Query}$  [ $\rho_0, \mathbf{info1}$ ];

```

```

LOCAL  $r_{c_2}$  = NEW[] SET;
LOCAL info2 = NEW[ $r_{c_2}$ ] CHAN;
RUN  $E_{Buy}[\rho_0, \mathbf{info2}]$ ;
DELETE [ $r_{c_2}$ ];
DELETE [ $r_{c_1}$ ]
}

```

The effect processes E_{Query} and E_{Buy} are scheduled concurrently and instantiated with the same state, ρ_o . With this model, mILCruncher detects an interleaving in which the effect process for the Buy method is scheduled before the effect process for the

5 Query method, leading to an assertion violation.

The asynchronous constructs of the disclosed programming language have many connections with previous languages. None of these previous languages, however, support contracts or provide techniques to statically check program behavior. The disclosed novel futures are inspired by futures in Multilisp. Futures have appeared in

10 many other languages, such as CML, where they are not primitive, but are implemented with CML events. The join construct of the disclosed programming language is inspired by the join calculus, and is also the basis of a similar construct in Polyphonic C#, another extension to C#. The disclosed joins differ in that they synchronize dynamic, first-class futures rather than statically declared method entries. The timeout feature disclosed

15 herein with respect to joins is similar to a construct in the Orc language.

The notion of contracts and algorithm for conformance checking in the disclosed programming language are related to a number of recent systems in software model checking, static analysis, and behavioral type systems. Systems such as SLAM, ESC Java, Vault, and ESP, check sequential specifications against sequential code. By

20 contrast, concurrency is a primitive for the disclosed language contract-checking algorithm. Moreover, several of these conventional tools rely on global alias analysis for sound model extraction, whereas the disclosed language uses region-and-effect analysis to extract its model and further refines the alias relationships at model-checking time. The Bandera toolkit model checks concurrent Java programs. Its model extraction is

25 based on program slicing and a user-supplied abstraction library. The disclosed model extraction algorithm uses type-and-effect inference and source-level tracked types.

Moreover, unlike Bandera, the disclosed programming language uses programmer-specified contracts to separately check client and server code.

The disclosed model extraction algorithm builds on previous work in type-and-effect systems, both for region-based memory management and for concurrent behavior analysis. The disclosed regions are used to control sound model extraction and for garbage collection in the model checker. In addition, the disclosed regions are re-interpreted as sets with a non-deterministic choice operator.

Programming languages are powerful tools for expressing and checking complex systems. The disclosed programming language provides an effective programming model for asynchronous computation and for performing lightweight, modular checking of concurrent control properties. The checking algorithm is a novel combination of techniques from type inference and model checking.

Referring now to FIG. 4, there is illustrated a general system implementation of the APL 100 of the present invention. A first client interface 400 (also denoted Client Interface₁) includes one or more clients (also denoted Client₁,...,Client_N) that interface with a first provider interface 402 (also denoted Provider Interface₁) via a network (not shown). The provider interface 400 includes one or more service providers (or services) (also denoted as Service₁,...,Service_N) that provide services to the clients. The first client interface 400 includes a first client 404 (also denoted Client₁) seeking services from a first service provider 406.

In accordance with the present invention, the disclosed APL 100 allows asynchronous interaction between the clients of the first client interface 400 and the services of the first service provider 402 by extracting models and, defining and enforcing contracts therebetween. Thus a first contract 408 (also denoted Contract₁) is generated between the first client 404 and the first service 406 by providing the capability to extract a corresponding model from each of the first client 404 and the first service 406, define the contract terms in accordance with asynchronous language, and enforce the first contract 408 with the conformance checker.

Similarly, where the first client interface 400 includes an Nth client 410 (also denoted Client_N) seeking services from the provider interface 402 that includes an Nth service 412 (also denoted Service_N), an Nth contract 414 (also denoted Contract_N) is

created and enforced. Of course, in network environments, there may be a plurality of individual client interfaces, such that an Nth client interface 416 (also denoted Client Interface_N) communicates with an Nth provider interface 418 (also denoted Provider Interface_N) to access services provided therein. Thus, a number of corresponding
5 contracts 420 may be created and enforced for asynchronous data exchange in accordance with the present invention.

Referring now to FIG. 5, there is illustrated an exemplary computing environment 500 in which the inventions may be implemented. The environment includes a computer 502, the computer 502 including a processing unit 504, a system memory 506 and a
10 system bus 508. The system bus 508 couples system components including, but not limited to the system memory 506 to the processing unit 504. The processing unit 504 may be any of various commercially available processors. Dual microprocessors and other multi-processor architectures also can be employed as the processing unit 504.

The system bus 508 can be any of several types of bus structure including a
15 memory bus or memory controller, a peripheral bus and a local bus using any of a variety of commercially available bus architectures. The system memory 506 includes read only memory (ROM) 510 and random access memory (RAM) 512. A basic input/output system (BIOS), containing the basic routines that help to transfer information between elements within the computer 502, such as during start-up, is stored in the ROM 510.

20 The computer 502 further includes a hard disk drive 514, a magnetic disk drive 516, (*e.g.*, to read from or write to a removable disk 518) and an optical disk drive 520, (*e.g.*, reading a CD-ROM disk 522 or to read from or write to other optical media). The hard disk drive 514, magnetic disk drive 516 and optical disk drive 520 can be connected to the system bus 508 by a hard disk drive interface 524, a magnetic disk drive interface
25 526 and an optical drive interface 528, respectively. The drives and their associated computer-readable media provide nonvolatile storage of data, data structures, computer-executable instructions, and so forth. For the computer 502, the drives and media accommodate the storage of broadcast programming in a suitable digital format. Although the description of computer-readable media above refers to a hard disk, a
30 removable magnetic disk and a CD, it should be appreciated by those skilled in the art that other types of media which are readable by a computer, such as zip drives, magnetic

cassettes, flash memory cards, digital video disks, cartridges, and the like, may also be used in the exemplary operating environment, and further that any such media may contain computer-executable instructions for performing the methods of the present invention.

5 A number of program modules can be stored in the drives and RAM 512, including an operating system 530, one or more application programs 532, other program modules 534 and program data 536. It is appreciated that the present invention can be implemented with various commercially available operating systems or combinations of operating systems.

10 A user can enter commands and information into the computer 502 through a keyboard 538 and a pointing device, such as a mouse 540. Other input devices (not shown) may include a microphone, an IR remote control, a joystick, a game pad, a satellite dish, a scanner, or the like. These and other input devices are often connected to the processing unit 504 through a serial port interface 542 that is coupled to the system
15 bus 508, but may be connected by other interfaces, such as a parallel port, a game port, a universal serial bus ("USB"), an IR interface, etc. A monitor 544 or other type of display device is also connected to the system bus 508 *via* an interface, such as a video adapter 546. In addition to the monitor 544, a computer typically includes other peripheral output devices (not shown), such as speakers, printers etc.

20 The computer 502 may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer(s) 548. The remote computer(s) 548 may be a workstation, a server computer, a router, a personal computer, portable computer, microprocessor-based entertainment appliance, a peer device or other common network node, and typically includes many or all of the elements
25 described relative to the computer 502, although, for purposes of brevity, only a memory storage device 550 is illustrated. The logical connections depicted include a LAN 552 and a WAN 554. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

 When used in a LAN networking environment, the computer 502 is connected to
30 the local network 552 through a network interface or adapter 556. When used in a WAN networking environment, the computer 502 typically includes a modem 558, or is

connected to a communications server on the LAN, or has other means for establishing communications over the WAN 554, such as the Internet. The modem 558, which may be internal or external, is connected to the system bus 508 *via* the serial port interface 542. In a networked environment, program modules depicted relative to the computer 502, or portions thereof, may be stored in the remote memory storage device 550. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

Referring now to FIG. 6, there is illustrated a schematic block diagram of a networking environment 600 in accordance with the present invention. The system 600 includes one or more client(s) 602. The client(s) 602 can be hardware and/or software (*e.g.*, threads, processes, computing devices). The client(s) 602 can house cookie(s) and/or associated contextual information by employing the present invention, for example. The system 600 also includes one or more server(s) 604. The server(s) 604 can also be hardware and/or software (*e.g.*, threads, processes, computing devices). The servers 604 can house threads to perform transformations by employing the present invention, for example. One possible communication between a client 602 and a server 604 may be in the form of a data packet adapted to be transmitted between two or more computer processes. The data packet may include a cookie and/or associated contextual information, for example. The system 600 includes a communication framework 606 that can be employed to facilitate communications between the client(s) 602 and the server(s) 604. The client(s) 602 are operably connected to one or more client data store(s) 608 that can be employed to store information local to the client(s) 602 (*e.g.*, cookie(s) and/or associated contextual information). Similarly, the server(s) 604 are operably connected to one or more server data store(s) 610 that can be employed to store information local to the servers 604.

What has been described above includes examples of the present invention. It is, of course, not possible to describe every conceivable combination of components or methodologies for purposes of describing the present invention, but one of ordinary skill in the art may recognize that many further combinations and permutations of the present invention are possible. Accordingly, the present invention is intended to embrace all such alterations, modifications and variations that fall within the spirit and scope of the

appended claims. Furthermore, to the extent that the term “includes” is used in either the detailed description or the claims, such term is intended to be inclusive in a manner similar to the term “comprising” as “comprising” is interpreted when employed as a transitional word in a claim.